AFRL-RI-RS-TR-2012-070

HIGH PERFORMANCE COMPUTING MULTICAST

CORNELL UNIVERSITY

FEBRUARY 2012

FINALTECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2012-070 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
GEORGE RAMSEYER
Work Unit Manager

/s/
PAUL ANTONIK, Technical Advisor
Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEB 2012	Final Technical Report	JUN 2010 – SEP 2011

4. TITLE AND SUBTITLE

HIGH PERFORMANCE COMPUTING MULTICAST

5a. CONTRACT NUMBER
FA8750-10-1-0181

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Kenneth Birman
Daniel Freedman
Robert van Renesse
Hakim Weatherspoon
Tudor Marian

5d. PROJECT NUMBER
T2DP

5e. TASK NUMBER
CO

5f. WORK UNIT NUMBER
RN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
Upson Hall, Room 4119B
341 Pine Tree Road
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-070

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. **PA# 88ABW-2012-0835**
Date Cleared: **15 FEB 2012**

13. SUPPLEMENTARY NOTES

14. ABSTRACT This investigation of High Performance Computing (HPC) Multicast for High-Speed Publication-Subscription (Pub-Sub) sought to deliver both insight into and implementation of high-performance multicast solutions that enable better utilization of cloud resources. These solutions combine improved scalability with increased consistency — ensuring that expected and necessary system conditions are thus met for a myriad of critical national-asset applications that are likely to move to the cloud in the next decade. In the context of this effort, the applicability of the oft-invoked Consistency, Availability and Partition tolerance (CAP) theorem was explored within specific environments of commonly deployed clouds, and novel insights into CAP's tradeoffs were developed between CAP and its conclusion that a replicated service can possess just two of the three. It was determined that there are replicated services for which the applicability of CAP is unclear — specifically, the scalable "soft-state" services that run in the first-tier of a single cloud-computing data center. The challenge is that such services live in a single data center and run on redundant networks. Partitioning events involve single machines or small groups, and are treated as node failures; thus, the CAP proof doesn't apply in a formal sense, as it's proven by forcing a replicated service to respond to conflicting requests during a partitioning failure, triggering inconsistency. Nonetheless, most developers believe in a generalized CAP "folk theorem," holding that scalability and elasticity are incompatible with strong forms of consistency. We designed, implemented, and benchmarked the Isis^2 platform: a first-tier consistency alternative that replicates data, combines agreement on update ordering with amnesia freedom, and supports both good scalability and fast response. A team of students was lead in the application of Isis^2 to build a large-scale distributed computer-vision landmark-recognition system, thus demonstrating the practicality of Isis^2 from a software-engineering perspective. Isis^2 is publically available, without patent or other intellectual-property encumbrances, via a 3-clause Berkeley Software Distribution (BSD) license.

15. SUBJECT TERMS

High Performance Computing, multicast, CAP Theorem, Isis, Publication-Subscription

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			GEORGE RAMSEYER
U	U	U	UU	33	19b. TELEPHONE NUMBER (*Include area code*) N/A

TABLE OF CONTENTS

LIST OF FIGURES

1. SUMMARY

This investigation of High Performance Computing (HPC) Multicast for High-Speed Publication-Subscription (Pub-Sub) sought to deliver both insight into and implementation of high-performance multicast solutions that enable better utilization of cloud resources. Our solutions combine improved scalability with increased consistency — ensuring that expected and necessary system conditions are thus met for a myriad of critical national-asset applications that are likely to move to the cloud in the next decade. In the context of this effort, the applicability of the oft-invoked Consistency, Availability and Partition tolerance (CAP) theorem was explored within specific environments of commonly deployed clouds, and novel insights into CAP's tradeoffs were developed between CAP and its conclusion that a replicated service can possess just two of the three. We discovered that there are replicated services for which the applicability of CAP is unclear, including the scalable "soft-state" services that run in the first-tier of a single cloud-computing data center. The puzzle is that such services live in a single data center and run on redundant networks. Partitioning events involve single machines or small groups and are treated as node failures; thus, the CAP proof doesn't apply in a formal sense, as it's proven by forcing a replicated service to respond to conflicting requests during a partitioning failure, triggering inconsistency. Nonetheless, most developers believe in a generalized CAP "folk theorem," holding that scalability and elasticity are incompatible with strong forms of consistency. We designed, implemented, and benchmarked the Isis^2 platform: a first-tier consistency alternative that replicates data, combines agreement on update ordering with amnesia freedom, and supports both good scalability and fast response. We have led a team of students in the application of Isis^2 to build a large-scale distributed computer-vision landmark-recognition system, thus demonstrating the practicality of Isis^2 from a software-engineering perspective. Isis^2 is publically available, without patent or other intellectual-property encumbrances, via a 3-clause Berkeley Software Distribution **(**BSD) license.

2. INTRODUCTION

Figure 1. Part of Cornell University's Computing Resources

The United States needs a new generation of cheaper high-performance computing systems even as technology trends are shifting towards inexpensive raw computing power: some of the world's fastest processors are now commodities used in gaming, multicore is becoming common, and advances in core Internet routing have helped optical switching leap into the 100Gbps range, with 1Tbps within sight. Thus the opportunity now exists to create HPC platforms that will cost a fraction of what previous generations of machines cost, and yet may actually outperform most existing solutions. In Fig. 1 is presented a portion of Cornell University's research facility, which focuses, in part, on research for high speed networks. In particular, the networks shown in Figure 2 have been well characterized.

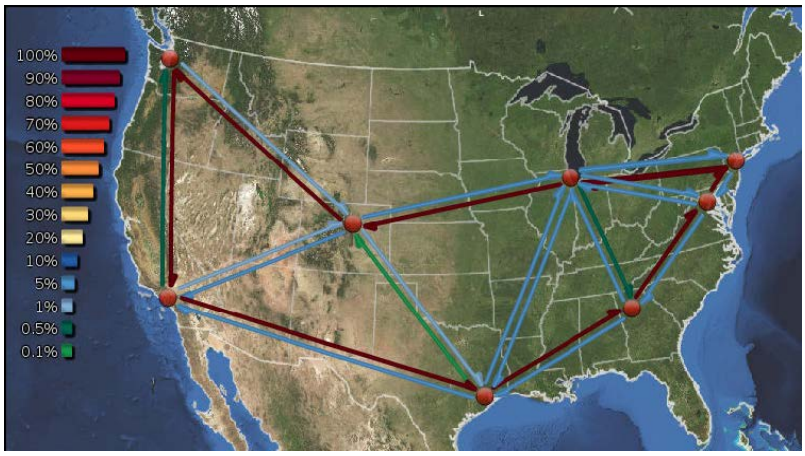

Figure 2. High Speed Networks

2.1 Serious Obstacles in Existing Solutions

While existing trends reveal a possible path forward, they also confront us with serious obstacles. HPC programming solutions can only scale to a limited degree on cluster-style computing elements. A core issue here is that technologies such as Parallel Virtual Machines (PVM) and Message-Passing Interface (MPI) aren't fault-tolerant, and this limitation was "baked in" so long ago that it simply can't easily be changed. Fault-tolerance is achieved by engaging in constant checkpoints (at significant cost), and because cluster-style computing systems typically experience crashes roughly in proportion to the number of nodes, the frequency of checkpoints needs to increase at that same rate. The amount of information being backed up will also rise as a function of the number of nodes. Thus we have (more or less) a quadratic scalability wall: beyond a certain size, HPC applications will need to be fragmented into multiple but loosely coupled applications, each small enough to make significant progress between checkpoints, and each working on a distinct aspect of the overall application.

Our vision of HPC applications as a collection of highly parallel components that exchange results in a loosely coupled manner encounters a different issue: HPC platform lack adequately fast tools for replicating data and exchanging "events" under demanding conditions. For example, the standard Enterprise Service Bus (ESB) model has frustratingly poor scalability properties.

Our research at Cornell reveals that a core problem with ESB performance is that for many reasons these systems are unable to take advantage of hardware IP multicast. Two important ones are these: First, modern HPC switching and routing components collapse if too many Internet Protocol (IP) multicast groups are in use; they start to deliver all messages to all network interface adapters, and those then become overloaded and drop packets. Further, multicast platforms can become unstable when using hardware multicast mechanisms, behaving in a bursty (oscillatory) manner. Jointly, these issues can cripple the ESB publish-subscribe technology.

2.2 Prior Work

In prior work, with Air Force research support (augmented by Intel and National Science Foundation (NSF) funding), our Cornell-based effort looked closely at some of these issues and found that the problems are explicable and, with appropriate software, can be brought under control. For example, in our paper at the 2010 ACM EuroSys conference [1], we showed that by appropriately "managing" the set of IP multicast addresses, one can avoid overloading the cluster router, eliminating the issue described above. Other work on our Quicksilver Scalable Multicast (award paper) [2] demonstrated that a collection of careful protocol design techniques can eliminate the bursty multicast instability.

As discussed above, this effort builds atop our earlier successes in a series of previous projects here at Cornell. Our work on methodical measurement of Wide-Area Networks (WANs) [3] prepared us to now examine data-center based Local-Area Networks (LANs) and

work to improve the HPC multicast implementations therein. Specifically, in prior work [4] we introduced a new methodology called BiFocals, with application to many aspects of protocol design and evaluation. This instrumentation has two components: one to send engineered bitstreams in which we precisely control the bits on the Physical Layer, and a second that can achieve similar precision in extracting timing and other statistics on the receive side. We employed BiFocals to perform the most accurate measurements of a WAN ever undertaken. Our findings refuted several common assumptions about network behavior.

The motivation for our BiFocals measurements stemmed from yet earlier work [5] that examined WAN's using typical software techniques and commodity endpoints. In that study, we undertook a careful examination of the end-to-end characteristics of an uncongested lambda network running at high speeds over long distances, identifying scenarios associated with loss, latency variations, and degraded throughput at attached end-hosts. We used identical fast commodity source and destination platforms, hence expect the destination to receive more or less what we send. We observed otherwise: degraded performance is common and easily provoked. In particular, the receiver loses packets even when the sender employs relatively low data rates. Data rates of future optical network components are projected to outpace clock speeds of commodity end-host processors, hence more and more end-to-end applications will confront the same issue we encounter. Our work thus posed a new challenge for those hoping to achieve dependable performance in higher-end networked settings.

Further, our past success with Dr. Multicast [1] provided the underlying IPMC solutions atop which we build the new Isis^2 platform, which we then demonstrated for HPC MC in the AFRL's Condor cluster. In our Dr. Multicast paper, we noted that IP Multicast (IPMC) in data centers becomes disruptive when the technology is used by a large number of groups, a capability desired by event notification systems. We traced the problem to root causes, and introduce Dr. Multicast (MCMD), a system that eliminates the issue by mapping IPMC operations to a combination of point-to-point unicast and traditional IPMC transmissions guaranteed to be safe. MCMD optimizes the use of IPMC addresses within a data center by merging similar multicast groups in a principled fashion, while simultaneously respecting hardware limits expressed through administrator-controlled policies. The system is fully transparent, making it backward-compatible with commodity hardware and software found in modern data centers. Experimental evaluation showed that MCMD allows a large number of IPMC groups to be used without disruption, restoring a powerful group communication primitive to its traditional role.

Finally, we've shown in prior work [2], recognized with a conference best paper award, that a careful and principled design of a multicast protocol can distinctly improve performance, as we hope to again try to do so with HPC multicast. QuickSilver Multicast (QSM) is a multicast engine designed to support a style of distributed programming in which application objects are replicated among clients and updated via multicast. The model requires platforms that scale in dimensions previously unexplored; in particular, to large numbers of multicast groups. Prior systems weren't optimized for such scenarios and can't take advantage of regular group overlap

patterns, a key feature of our application domain. Furthermore, little is known about performance and scalability of such systems in modern managed environments. We shed light on these issues and offer architectural insights based on our experience building QSM.

2.3 Application to Military Needs

One might reasonably inquire into the connection between the results we provide in this effort and the larger needs of the military and its warfighters. The scalability and consistency deliverables of our Isis^2 platform, along with an accompanying understanding of the scenarios to which it is applicable, serve as key contributions of this effort. As we discuss below, we tailor Isis^2 to the soft-state first-tier of the cloud, thus providing application architects with the ability to deliver scalable, consistent interactions with remote clients. Now, many existing applications that utilize the cloud infrastructure to focus on consumer, or even business, needs do not require higher levels of consistency, or, if they do, they can willingly trade performance to meet such requirements.

However, as Department of Defense (DoD) information-system infrastructure transitions to the cloud, it will continue to interface with forward-deployed Command and Control (C2) clients. Operating in a fluid real-time environment, these clients will be unwilling to sacrifice performance, and yet will clearly require levels of consistency beyond that often provided in the first tier of a cloud computing environment. The type of eventual consistency that has become the coin of the realm for the Amazons and the eBays of the world is much less compelling when the problem space involves tactical situation awareness, or deconfliction of units or airspace, or a variety of other critical military operational uses, such as those motivated by Figure 3.

It is therefore within this context, and ones similar to it, that the strength of our agenda here is best recognized. We have succeeded in delivering scalable consistent solutions that tackle portions of the design space that have largely been overlooked by the wider research community.

Further, this agenda builds naturally from our previous AFRL-sponsored effort which created the Live Objects middleware and delivered composable mashups to combine edge content with hosted content. That earlier effort considered the technical provisioning of such hosted content to be outside its scope. Now, in this AFRL-funded work, we tackle just such a problem and find a host of intriguing perspectives.

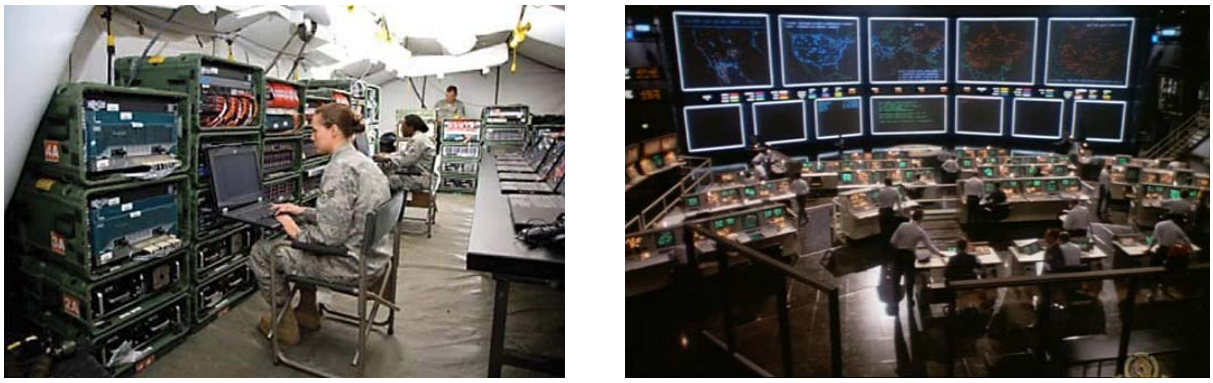

Figure 3. DoD Command and Control

C2 clients (left) interfacing with central DoD Command Center (right) motivated our effort. Edge clients in such an environment require higher levels of performance, as well as stronger guarantees of consistency, than those typically provided now by cloud infrastructures.

3. METHODS, ASSUMPTIONS AND PROCEDURES

The current effort focused on an attack on ESB scalability. In particular, there were four core research objectives, all of which were accomplished over the course of the one-year research program.

We are convinced that the key to successfully exploiting hardware multicast in HPC systems is to start by fully characterizing the behavior of hardware multicast in the target setting. While this may seem like a trivial observation, measurement is notoriously hard on modern computer systems: the optical fiber is first handled by hardware in the network interface card, then copied to memory associated with one of the multicore CPUs in the power PC itself, and then the driver, Operating System (O/S) and ESB (pub-sub application) all need to take actions before a multicast actually reaches the end-user's code. Loss can and does occur at every step and measurement is particularly difficult when the measurement code runs on the same platform that experiences the loss. We've completed development of a new instrumentation technology to this end, and in using it, we have achieved a dramatic improvement in the quality of our analysis of where loss occurs on this complex path, and precisely what conditions can trigger it. We combined this with an examination of the larger system-wide contributions to these conditions, and have leveraged this to a greater understanding of the design lessons that will continue to inform the architecture of network-attached endpoints and the intermediary switches and routers.

We designed and implemented a large-scale platform, which we call Isis^2, to allow increased scalability, availability, and performance on cloud resources. Isis^2 supports consistent, locally responsive cloud services. Responses to client requests can be computed using purely local data, hence delays are limited only by local computational costs. Updates propagate asynchronously and map to a single IP multicast; locking is usually avoided by employing primary-copy replication, and otherwise is performed with an inexpensive token-passing scheme. The approach relaxes durability for soft-state updates: in analog to the database community, this yields an "ACI and mostly D" model. Durability violations are concealed using a form of firewall.

We exhaustively benchmarked and tuned the Isis^2 system on the AFRL's production Condor cluster, investigating its ability to scale using physical IPMC, and the benefits derived from Isis^2's foundation upon our earlier Dr. Multicast work [1]. We explored the tradeoffs between different protocol designs and gained insights into the applicability of such protocols in different layers of the typical cloud and for different application architecture needs [6].

We explored the applicability of the oft-invoked Consistency, Availability and Partition tolerance (CAP) theorem to the very specific environmental details of commonly deployed clouds, and discovered important, and somewhat heretical, insights. CAP explores tradeoffs between its three constituents, and the theorem concludes that a replicated service can possess just two of the three. The theorem is proved by forcing a replicated service to respond to

conflicting requests during a partitioning failure, triggering inconsistency. However, there are replicated services for which the applicability of CAP is unclear. As part of our effort, we looked at scalable "soft-state" services that run in the first-tier of a single cloud-computing data center. The puzzle is that such services live in a single data center and run on redundant networks. Partitioning events involve single machines or small groups and are treated as node failures; thus, the CAP proof doesn't apply. Nonetheless, developers believe in a generalized CAP "folk theorem," holding that scalability and elasticity are incompatible with strong forms of consistency. We present a first-tier consistency alternative that replicates data, combines agreement on update ordering with amnesia freedom, and supports both good scalability and fast response.

While involving a large team of Masters students, we designed and built a sophisticated application atop Isis^2. This served as a critical proof-of-concept that the Isis^2 platform not only provides performance and consistency benefits in the abstract as a middleware solution, but can also be efficiently utilized, in terms of practical software engineering, by a team of junior programmers without much prior experience in distributed systems development, and with no exposure to the platform. Over a period of ten weeks, our team of four students transformed an existing computer vision research application, designed strictly for sequential uniprocessor execution, into a scalable, distributed application running atop Isis^2 on a cluster of nodes in a cloud.

As part of this effort, we have released the Isis^2 platform to the public under a (3-clause) BSD license with no patent or other restrictions. Our publications document the hard problems we solved, and they represent a roadmap that others could follow to create similar platforms that obtain greater consistency and performance on the critical first tier of the cloud.

4. RESULTS AND DISCUSSION

The CAP theorem [7][8] has been influential within the cloud computing community, and motivates the creation of cloud services with weak consistency properties. While the theorem focuses on services that span partition-prone network links, CAP is also cited in connection with Basically Available replicated Soft state with Eventual consistency (BASE), a methodology in which services that run in a *single* data center on a reliable network are engineered to use a non-transactional coding style, tolerate potentially stale or incorrect data, and eschew synchronization. eBay invented the approach [9], and Amazon points to the self-repair mechanisms in the Dynamo key-value store as an example of how eventual consistency behaves in practice [10].

The "first-tier" of the cloud, where this issue is prominent, is in many ways an unusual environment. When an incoming client's request is received, *fast response* is the overwhelming priority, even to the extent that other properties might need to be weakened. Cloud systems host all sorts of subsystems with strong consistency guarantees, including databases and scalable global file systems, but they reside "deeper" in the cloud, shielded from the heaviest loads by the first-tier.

To promote faster responsiveness, first-tier applications often implement replicated in-memory key-value stores, using them to store state or to cache data from services deeper in the cloud. When this data is accessed while processing a client request, locking is avoided, as are requests to inner services, for example to check that cached data isn't stale. To the extent that requests have side-effects that require updates to the cloud state, these are handled in a staged manner: the service member performs the update locally and responds to the client. Meanwhile, in the background (asynchronously), updates are propagated to other replicas and, if needed, to inner-tier services that hold definitive state. Any errors are detected and cleaned up later, hopefully in ways external clients won't notice: *eventual consistency*.

This works well for eBay, Facebook and other major cloud providers today. But will it work for tomorrow's cloud applications? For example, as applications such as medical records management (including "active" medical applications that provide outpatient monitoring and control), transportation systems (smart highways), control of the emerging smart power grid, and similar tasks shift to the cloud, we'll confront a wave of first-tier applications that may be required to justify their actions. Is a scalable solution to this problem feasible?

Our approach mimics many aspects of today's BASE solutions, but offers first-tier service replicas a way to maintain consistent replicated data, across the copies. Locks are not required: any service instance sees a consistent snapshot of the first-tier state, using locks only if a mutual exclusion property is required. To maximize performance, updates are performed optimistically and asynchronously, but to avoid the risk that updates might be lost in a later crash, we introduce

a *flush barrier*: before replying to an external client (even for a read-only operation), the primary checks to make sure that any unstable updates on which the request depends have reached all the relevant replicas. Jointly, these techniques yield a strong consistency guarantee and *amnesia freedom:* in-memory durability. We believe this is a good match to the performance, scalability and responsiveness needs of first-tier cloud services.

This effort doesn't have the scope to consider other alternatives, such as full-fledged transactions. Nonetheless, our experimental work supports the view that full-fledged atomic multicast wouldn't scale well enough for use in this setting (i.e. durable versions of Paxos[1], or Atomicity, Consistency, Isolation and Durability (ACID) transactions). As we'll show, strongly durable multicast exhibits marked performance degradation in larger-scale settings. In contrast, amnesia freedom scales well, overcoming the limitations of CAP.

4.1 Life in the First-Tier

Cloud-computing systems are generally structured into tiers: a first-tier that handles incoming client requests (from browsers, applications using web-services standards, etc), caches and key-value stores that run near the first-tier, and inner-tier services that provide database and file-system functionality. A wide variety of back-end applications run off the critical path, preparing indices and other data for later use by online services.

In the introduction some aspects of the first-tier programming model were discussed: aggressive replication, very loose coupling between replicas, optimistic local computation without locking, and asynchronous updates. Modern cloud-development platforms standardize this model, and in fact take it even further. In support of elasticity, first-tier applications are also required to be *stateless*: cloud platforms launch each new instance in a standard initial state, and they discard local data when an instance fails or is halted. These terms require some explanation. "Stateless" doesn't mean that these instances have no local data but rather that they are limited to non-durable *soft state*. On launch, a new instance initializes itself by copying data from some operational instance, or by querying services residing deeper in the cloud. Further, "elasticity" doesn't mean that a service might be completely shut down without warning: cloud-management platforms can keep some minimum number of replicas of each service running (ensuring continuous availability). Subject to these constraints, however, replication degree can vary rapidly.

These observations enable a form of durability. Data replicated within the soft state of a service, in members that the management platform won't shut down (because they reside within the core replica set), will remain available unless a serious failure causes all the replicas to crash simultaneously. Failures of that kind will be rare in a well-designed application; we'll leverage this observation below.

[1] Paxos is a family of protocols for solving consensus in a network of unreliable processors. Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communication medium may experience failures

This, then, is the context within which CAP is often cited as a generalized principle. In support of extremely rapid first-tier responses and fault-tolerance, developers have opted for relaxed consistency. This effort will shows that one can achieve similar responsiveness while delivering stronger consistency guarantees by adopting a consistency model better matched to the characteristics of the first-tier. The approach could enable cloud-hosting of applications that need strong justification for any responses they provide to users. Consistency can also enhance security: a security system that bases authorization decisions on potentially stale or incorrect underlying data is at risk of mistakes that a system using consistent data won't make.

4.2 Consistency: A Multi-Dimensional Property

Terms like *consistency* can be defined in many ways. In prior work on CAP, the "C" is defined by citing the database ACID model; the "C" in CAP is defined to be the "C" and "D" from ACID. Consistency, in effect, is conflated with durability. Underscoring this point, several CAP and BASE papers also point to Paxos [11], an atomic multicast protocol that provides total ordering and durability.

Durability is the guarantee that if an update has been performed, it will never be lost. Normally, the property is expected to apply even if an entire service crashes and then restarts. But notice that for a first-tier service, durability in this strongest sense conflicts with the soft-state limitation. By focusing on techniques that guarantee durability and are often used in hard-state settings, one risks reaching conclusions that relate to features not needed by first-tier services. With this in mind let's review other common but debatable assumptions:

4.2.1 Membership.

Any replication scheme needs a *membership model.* Consider some piece of replicated data in a first-tier service: the data might be replicated across the full set of first-tier application instances, or it might live just within some small subset of them (in the latter case the term *shard* is often used). Which nodes are supposed to participate?

For the case in which every replica has a copy of the data item, the answer is evident: all the replicas currently running. But notice that because cloud platforms vary this set elastically, the actual collection will change over time, perhaps rapidly. Full replication forces us to track the set, to have a policy for initializing a newly launched service instance, and to ensure that each update reaches all the replicas, even if that set is large.

For shard data any given item will be replicated at just a few members, hence a mapping from key (item-id) to shard is needed. Since each service instance belongs to just a few shards but potentially needs access to all of them, a mechanism is also needed whereby any instance can issue read or update requests to any shard. Moreover, since shard membership can change, we'll need to factor membership dynamics into the model.

One way to handle such issues is seen in Amazon's Dynamo key-value store [12], which is a form of Distributed Hash Table (DHT). Each node in Dynamo is mapped (using a hashing

function) to a location on a virtual ring, and the key associated with each item is similarly mapped to the ring. The closest node with a mapped id less than or equal to that of the item is designated as its primary owner, and the value is replicated to the primary and to the next few (typically three) nodes along the ring: the shard for that key. Shard mappings change as nodes join and leave the ring, and data is moved around accordingly (a form of *state transfer).* Coordination with the cloud-management service minimizes abrupt elasticity decisions that would shut down shards without first letting members transfer state to new owners.

A second way to implement shards arises in systems that work with *process groups* [14]: here, the various requirements are solved by a group communication infrastructure (such as new Isis^2 system). Systems of this sort offer an Application Programming Interface (API) with basic functionality: ways for processes to create, join, and leave groups; group names that might encode a key (such as *"shard123"*), a state-transfer mechanism to initialize a joining member from the state of members already active, and built-in synchronization (Isis^2, for example, implements the virtual synchrony model [14]). The developer decides how shards should work, then uses the provided API to implement the desired policy. With group communication systems, group membership change is a potentially costly event, but a single membership update can potentially cover many changes. Accordingly, use of this approach presumes some coordination with the cloud management infrastructure so that changes are done in batches. Assuming that the service isn't buggy, the remaining rate of failures should be very low.

A third shard-implementation option is seen in services that run on a stable set of nodes for a long period, enabling a kind of *static* membership in which some set of nodes is designated as running the service. Here, membership remains fixed, but some nodes may be down when a request is issued. This forces the use of quorum replication schemes, in which only a quorum of replicas see each update, but reading data requires accessing multiple replicas. State transfer isn't needed unless the static membership is reconfigured.

Several CAP papers express concern about the high cost of quorum operations, especially if they occur on the critical path for end-user request processing. Notice that quorum operations are needed only in the static membership case (not for DHT or process group approaches). Those avoid the need for quorums because they evolve shard membership as nodes join, leave or fail. This avoids a kind of non-local interaction that can be as costly as locking: if every read or update operation on the critical path entails interaction with multiple nodes, the reply to the end-user could be delayed by a substantial number of multi-node protocol events (because many requests will perform multiple reads and updates). With dynamic membership, we gain the ability to do reads and writes *locally* at the price of more frequent group membership updates.

4.2.2 Update Ordering.

A second dimension of consistency concerns the policy whereby updates are applied to replicas. A *consistent* replication scheme is one that applies the same updates to every replica in

the same order, and that specifies the correct way to initialize new members, or nodes recovering from a failure [13][14][6].

Update ordering costs depend on the pattern whereby updates are issued. In many systems, each data item has a primary copy through which updates are routed. Some systems shift the role of being primary around, but the basic idea is the same: in both cases, by delivering updates in the order they occur at the primary, without gaps, the system can be kept consistent across multiple replicas. The required multicast ordering mechanism is simple and very inexpensive.

A more costly multicast-ordering need arises if *every* replica can initiate concurrent, conflicting updates to the same data items. When concurrent updates are permitted, the multicast mechanism must select an agreed-upon order, at which point the delivery order can be used to apply the updates in a consistent order at each of the replicas. This is relevant only because the CAP and BASE point to protocols that do permit concurrent updates. Thus by requiring replicated data to have a primary copy, we can achieve a significant cost reduction.

4.2.3 Durability.

The third dimension involves durability of updates. Obviously, an update that has been performed is durable if the service doesn't forget it. But precisely what does it mean to have "performed" an update? And must the durability mechanism retain data across complete shutdowns of the full membership of a service or shard?

In applications where the goal is to replicate a database or file (some form of external storage), durability involves mechanisms such as write-ahead logs: all the replicas would push updates to their respective logs, then acknowledge that they are ready to commit the update, and then in a second phase, the updates in the logs can be applied to the actual database. Lamport's Paxos protocol [15][11] doesn't talk about the application per-se, but most implementations of Paxos incorporate this sort of logging of pending updates. This can be called *strong* durability, and it presumes a durable storage that will survive failures.

Recall that first-tier services are required to be stateless. Can a first-tier service replicate data in a way that offers a meaningful durability property? The obvious possibility is in-memory update replication: we could distinguish between a service that might respond to a client *before* every replica knows of the updates triggered by that client's request, and a service that delays until *after* every replica has acknowledged the relevant updates. If we call the former solution *non-durable* (if the service has *n* members, even a single failure can leave *n-1* replicas in a state where they will never see the update), what should we call this other solution? This will be the case we're referring to as *amnesia freedom:* the service won't forget the update unless all *n* members fail (as noted earlier, that will be rare). Notice that with amnesia freedom, any subsequent requests issued by the client, after seeing a response to a first request, will be handled by service instances "aware" of the updates triggered by that first request.

Amnesia freedom isn't perfect. If a serious failure does force an entire service or shard to shut down, unless the associated data is backed up on some inner-tier service, state will be lost.

But, if such events are rare, the risk may be acceptable. For example, suppose that applications for monitoring and controlling embedded systems such as medical monitoring devices move to cloud-hosted settings. While these roles do require consistency and other assurance properties, the role of monitoring is a continuous online one. Moreover, applications of this sort generally revert to a fail-safe mode when active control is lost. Here, one might not need any inner-tier service at all.

Applications that push updates to an inner service have a choice: they can wait for the update to be acknowledged, or could adopt amnesia freedom, but in so doing, accept a window of vulnerability for the period between when the update is fully replicated in the memory of the first-tier service, until it reaches the inner-tier. Another database analogy comes to mind: database mirroring is often done by asynchronously streaming a log, despite the small risk that a failure could cause updates to be lost. An amnesia-free approach has an analogously small risk: having replicated an update in the memory of a first-tier service, the odds of it being lost will be orders of magnitude smaller than in today's BASE approaches.

4.2.4 Failure Model.

This effort assumes that applications fail by crashing, and that network packets can be lost, and that partitioning failures that isolate a node, or even a rack or container are mapped to crash failures: when the network is repaired, the nodes that had been isolated will be forced to restart. We also assume that while isolated by a network outage, nodes are unable to communicate with external clients.

4.2.5 Putting It All Together.

We arrive at a rather complex set of choices and options, from which one can construct a diversity of replication solutions with very different properties, required structure, and expected performance. Some make little sense in the first-tier; others represent reasonable options:

One can build protocols that replicate data optimistically and later heal any problems that arise, perhaps using gossip (BASE). Updates are applied in the first-tier, but then passed to inner-tier services which might apply them in different orders.

One can build protocols synchronized with respect to membership changes, and with a variety of ordering and durability properties (virtual synchrony and also "in-memory" versions of Paxos, where the Paxos durability guarantee applies only to in-memory data). Amnesia freedom is achieved by enforcing a "barrier": prior to sending a reply to the client request, the system pauses, delaying the response until any updates initiated by the request (or seen by the request through its reads) have reached all the replicas and thus become stable. If all updates are already stable, no delay is incurred.

One can implement a strongly-durable state-machine replication model. Most implementations of Paxos use this model. In our target scenario, of course, the strongest forms

of durability just aren't meaningful: there is no benefit to logging messages other than in the memory of the replicas themselves.

One can implement database transactions in the first-tier, coupling them to the serialization order used by inner-tiers, for example via a true multi-copy model based on the ACID model or a *snapshot-isolation* model. This approach is receiving quite a bit of attention in the research community.

How does CAP deal with this diversity of options? The question is easier to pose than to answer. Brewer's 2000 Principles of Distributed Computing (PODC) keynote [7] proposed CAP as a general principle. His references to consistency evoke ACID database properties. Gilbert and Lynch offered their proof [8] in settings with partitionable wide-area links, and in fact pointed out that with even slight changes, CAP ceases to apply (for example, they propose a model called *t-eventual consistency* that avoids the CAP tradeoff). In their work on BASE, Pritchett [9] and Vogels [10] point both to the ACID model and to the durable form of Paxos [15][11]. They argue that these models will be too slow for the first-tier; their concerns apparently stem from the costly two-phase structure of these particular protocols, and from their use of quorum reads and updates, resulting in delays on the critical path that computes responses.

Notice that the performance and scalability concerns in question stem from durability mechanisms, not those supporting order-based consistency. As we saw earlier, sharded data predominates in the first-tier, and one can easily designate a primary copy at which updates are performed first. Other replicas mirror the primary. Thus the "cost of consistency" can be reduced to the trivial requirement that updates be performed in the same FIFO order used by the primary.

We recommend that first-tier services employ a shared model, with a primary replica for each data item, and we favor a process-group model that coordinates group membership changes with updates: virtual synchrony [14]. When a first-tier service instance receives a request, it executes it using local data, and applies any updates locally as well, issuing a s**t**ream of asynchronous updates that will be delivered and applied in First In, First Out (FIFO) order by other replicas. This permits a rapid but optimistic computation of the response to the user: optimistic not because any rollback might be needed, but because failure could erase the resulting state. This risk is eliminated by imposing a synchronization barrier prior to responding to the client, even for read-only requests. The barrier delays the response until any prior updates have become stable, yielding a model with strong consistency and amnesia freedom. Relative to today's first-tier model, the only delay is that associated with the barrier event.

4.3 The Isis^2 System

In this section, we offer a brief experimental evaluation of the system, focused on the costs of our scheme. Our new Isis^2 system supports virtually synchronous process groups and includes reliable multicasts with various ordering options. The **Send** primitive is per-sender FIFO ordered. An **OrderedSend** primitive guarantees total order; we won't be using it here because

we're assuming that sharded data has a primary copy. The barrier primitive is called **Flush;** it waits passively until prior multicasts become stable.

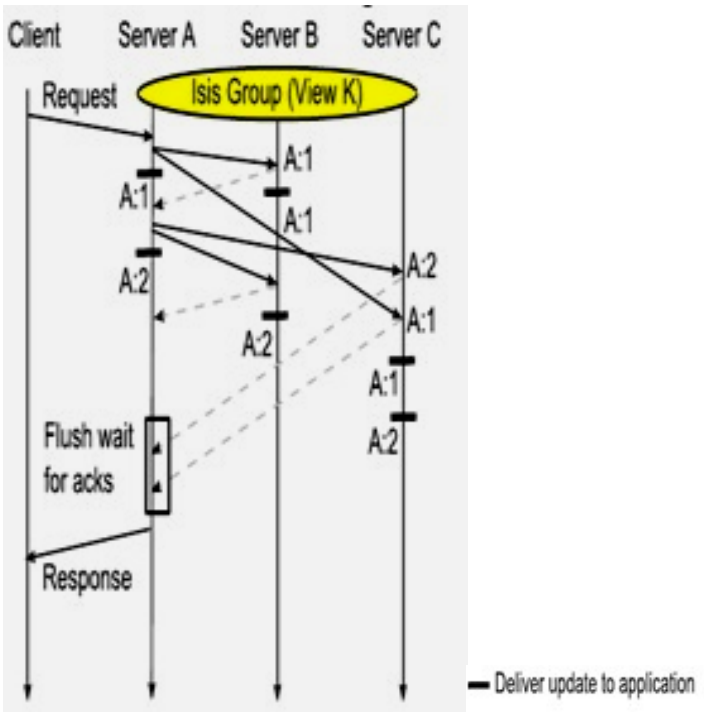

Figure 4. Send, together with a Flush barrier.

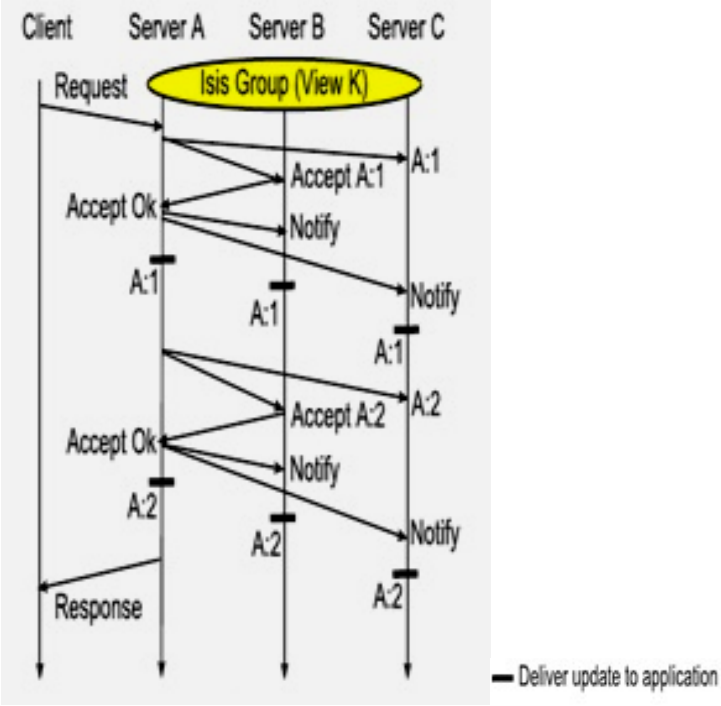

Figure 5. SafeSend (in-memory) Paxos.

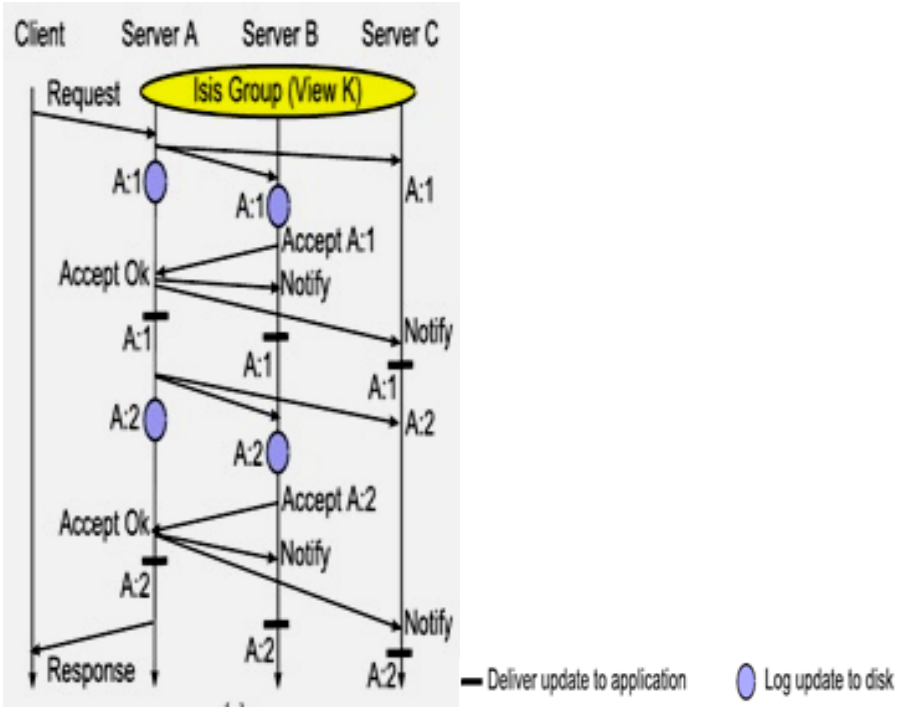

Figure 6. Durable (disk-logged) Paxos.

Isis[2] also supports a virtually synchronous version of Paxos [14], via a primitive we call **SafeSend**. The user can specify the size of the acceptor set; we favor the use of three acceptors,

but one could certainly select all members in a process group to serve as acceptors, or some other threshold appropriate to the application. **SafeSend** offers two forms of durability: in-memory durability, which we use for soft-state replication in the first-tier, and true on-disk durability. Here, we only evaluate the in-memory configuration.

In Figures 4-6 are illustrated these protocol options. A:1 and A:2 are two updates sent from Server A. These runs all use a single-threaded sender. With multiple threads **SafeSend** would have a more overlapped pattern of traffic that wouldn't impact the per-invocation latency metrics on which our evaluations focuses. Our key insight is that for soft-state replication, **SafeSend** is no stronger than that of **Send**+**Flush**.

In Figure 4 we see an application that issues a series of **Send** operations and then invokes **Flush,** which causes a delay until all the prior **Sends** have been acknowledged. In this particular run, updates A:1 and A:2 arrive out of FIFO order at member C, which delays A:2 until A:1 has been received; we illustrate this case just to emphasize that FIFO ordering is needed, but inexpensive to implement. To avoid clutter, Figure 4 omits stability messages that normally piggyback on outgoing multicasts: they inform the replicas that prior multicasts have become stable. Thus, some future update, A:3, might also tell the replicas that A:1 and A:2 are stable and can be garbage-collected.

Figure 5 shows our in-memory Paxos protocol with 2 acceptors (nodes A and B) and one additional member (node C); all three are *learners* (all deliver messages to the application layer). Figure 6 shows this same case, but now with a durable version of the Paxos protocol: pending requests are now logged to disk rather than being held in memory. Not shown is the logic by which the log files are later garbage collected; it requires additional rounds of agreement, but they can be performed offline.

Notice that had we responded to the client in Figure 4 without first invoking the **Flush** barrier, even a single failure could cause amnesia; once **Flush** completes, the client response can safely be sent. **Flush** is needed even for a read-only client-request, since a read might observe data that was recently updated by a multicast that is still unstable. On the other hand, a **Flush** would be free unless an unstable multicast is pending. In fact, our experiments consider the worst-case: a pure-update workload; even so, **Flush** turns out to be cheap. For a read-intensive workload, it would be cost even less, since there would often be no unstable multicasts, hence no work to do.

The experiment we performed focuses on the performance-limiting step in our consistent-replication scheme. In many papers on data replication, update throughput would be the key metric. For our work, it is more important to measure the *latency to perform remote updates*, relative to the responsiveness goals articulated earlier. The reasons are dual. On the sender side (e.g. Server A in Figure 4), all costs are local until the call to **Flush**, and the delay for that call will be determined by the latency of the earlier asynchronous **Send** operations. For **SafeSend**,

the latency until delivery to the sender and to the other receivers is even more crucial; the sender cannot do local updates until a sufficient set of acceptors acknowledges the first-phase request. Thus, the single metric that tells us the most about the responsiveness of our solution is latency from when updates are sent, to when they are received and the application is able to perform them. We decided not to introduce membership churn; if we had done so, membership changes would have brought additional group reconfiguration overheads. Isis^2 batches membership changes, applying many at a time. The protocol normally completes in a few milliseconds, hence even significant elasticity events can be handled efficiently.

We ran our experiment on 48 nodes, each with twelve Xeon X5650 cores and connected by Gigabit Ethernet. Experiments with group sizes up to 48 used one member per node; for larger configurations we ran twelve members per-node. We designed a client to trigger bursts of work during which five multicasts are initiated (e.g. to update five replicated data objects). Two cases are considered: for one, the multicasts use **Send** followed by a **Flush;** the other uses five calls to **SafeSend,** in its in-memory configuration with no flush. Figure 4 reports the latency between when processing started at the leader and when update delivery occurred, on a per-update basis. All three of these consistent-replication options scale far better than one might have expected on the basis of the literature discussed earlier, but the amnesia-free **Send** significantly outperforms **SafeSend** even when configured with just three acceptors.

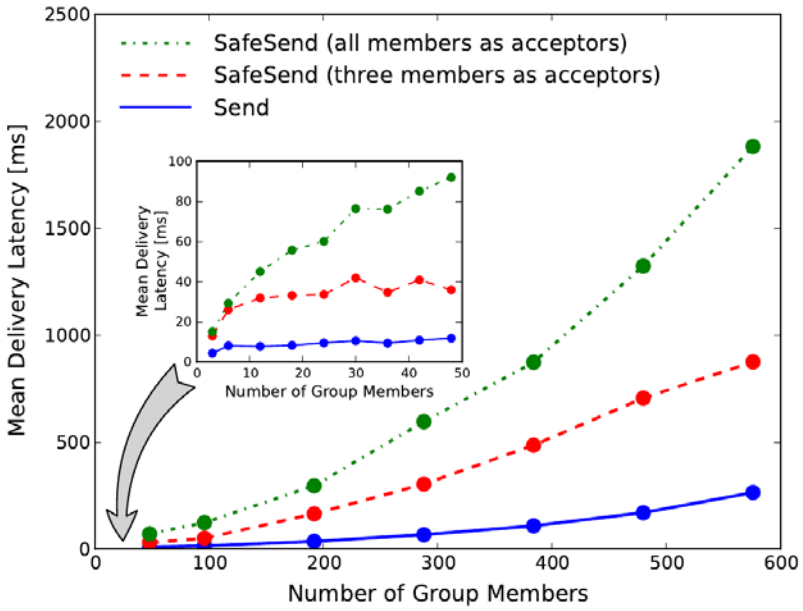

Figure 7. Mean delivery latency per single invocations.

Send primitive, contrasted with the same metric for **SafeSend** with three acceptors and **SafeSend** with acceptors equal to the size of the group.

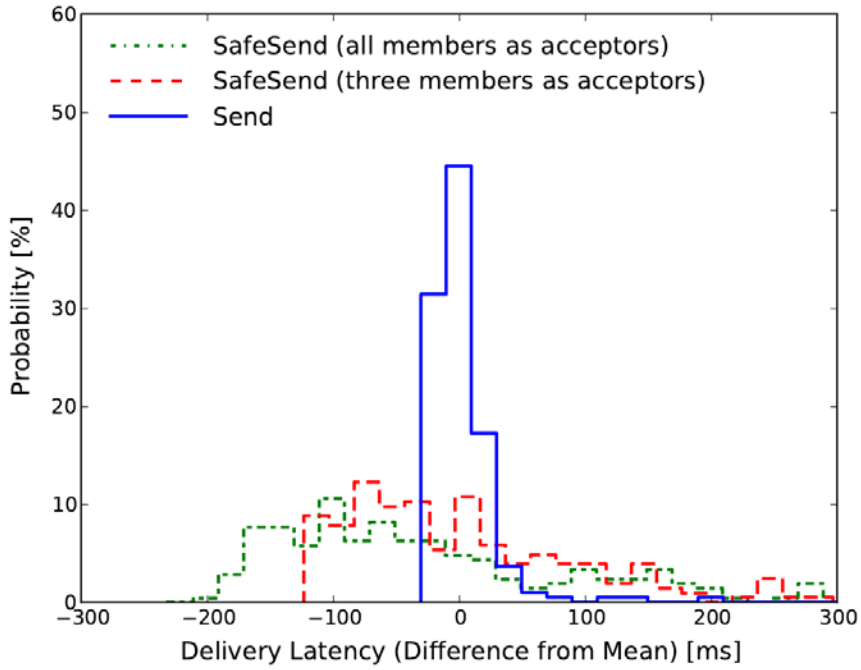

Figure 8. Histogram of delivery jitter.

Variance in latency relative to the mean, for all three protocols in a group size of 192 members.

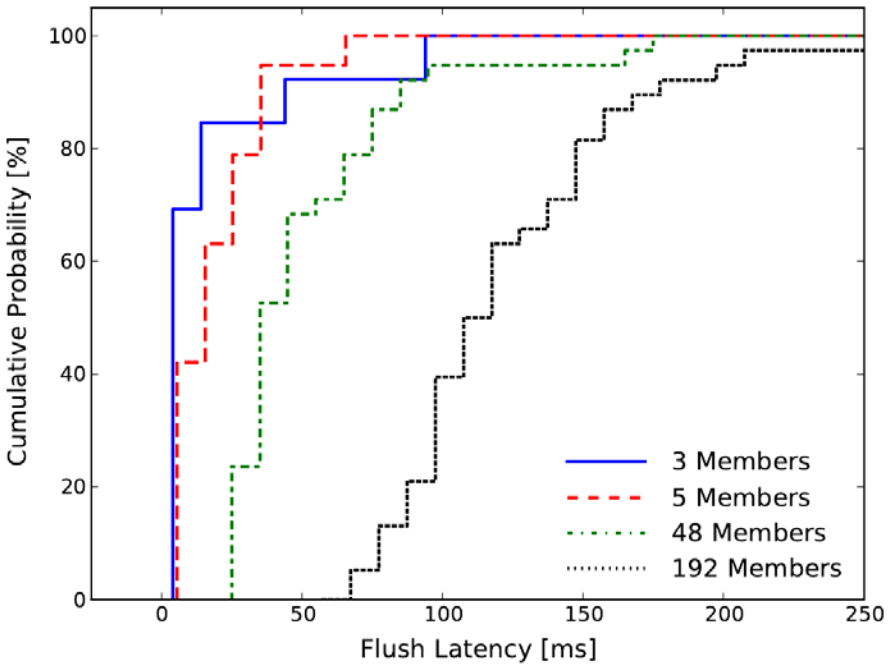

Figure 9. Cumulative Distribution Function (CDF) of delays.

The CDF's are associated with the Flush barrier.

Our experiments looked at replication on two scales. Sharded data is normally replicated in fairly small groups; we assumed these would have a minimum of 3 and a maximum of 48 members (inset in Figure 7). Then we scaled out more aggressively (placing members on each core of each physical node) to examine groups with as many as 576 members. We limited ourselves to evaluating **SafeSend** in its in-memory configuration (Figure 8), leaving studies of durable-mode performance (Figure 9) for future investigation; this decision reflects our emphasis on soft-state services in the first-tier.

For Figure 8 we picked one group size, 192 members, and explored variance of delivery latency from its mean, as shown in Figure 7. We see that while **Send** latencies are sharply peaked around the mean, the protocol does have a tail extending to as much as 100ms but impacting just a small percentage of multicasts. For **SafeSend** the latency deviation is both larger and more common**.** These observations reflect packet loss: In loss-prone environments (cloud-computing networks are notoriously prone to overload and packet loss) each protocol stage can drop messages, which must then be retransmitted. The number of stages explains the degree of spread: **SafeSend** latencies spread broadly, reflecting instances that incur zero, one, two or even three packet drops (see Figure 8). In contrast, **Send** has just a single phase (Figure 7), and hence is at risk of at most loss-driven delay. Moreover, **Send** generates less traffic, and this results in a lower loss rate at the receivers.

The software releases developed here can be downloaded from http://www.cs.cornell.edu/ken/isis2/.

4.4 Related Work

There has been debate around CAP and the possible tradeoffs (CA/CP/AP). Our treatment focuses on CAP as portrayed by Brewer [7] and by those who advocate for BASE [9][10]. Other relevant analyses include Gray's classic analysis of ACID scalability [16], Wada's study of NoSQL consistency options and costs [17], and Kossman's [18][19][20] and Abadi's 6 discussions of this topic. Database research that relaxes consistency to improve scalability includes the Escrow transaction model [22], PNUTS [23], and Sagas [24]. At the other end of the spectrum, notable cloud services that scale well and yet offer strong consistency include GFS [25], BigTable [26] and Zookeeper [27]. Papers focused on performance of Paxos include the Ring-Paxos protocol [28][29] and the Gaios storage system [30].

Our own work employs a model that unifies Paxos (state-machine replication) with virtual synchrony [14]; in particular, the discussion of amnesia freedom builds on one in [14]. Other mechanisms that we've exploited in Isis^2 include the IPMC allocation scheme from Dr. Multicast [1], and the tree-structured acknowledgements used in QuickSilver Scalable Multicast [2].

5. CONCLUSIONS

CAP is centered on concerns that the ACID database model and the standard durable form of Paxos introduce unavoidable delays. Focusing carefully on the consistency and durability needs of first-tier cloud services, it was shown here that strong consistency and a form of durability we call amnesia freedom can be achieved with very similar scalability and performance to today's first-tier methodologies. Our approach would also be applicable elsewhere in the cloud.

Obviously, not all applications need the strongest guarantees, and perhaps this is the real insight. Today's cloud systems are inconsistent by design because this design point works well for the applications that earn the revenue in today's cloud. The kinds of applications that need stronger assurance properties simply haven't wielded enough market power to shift the balance. The good news, however, is that if cloud vendors decide to tackle high-assurance cloud computing, CAP will not represent a fundamental barrier to progress.

6. REFFERENCES

[1] Vigfusson, Y., Abu-Libdeh, H., Balakrishnan, M., Birman, K., Burgess, R., Li, H., Chockler, G., and Tock, Y., "Dr. Multicast: Rx for Data Center Communication Scalability," *Proceedings of the 5th European Conference on Computer Systems (Eurosys '10)*, Paris, France, April 13-16, 2010.

[2] Ostrowski, K., Birman, K. and Dolev, D., "QuickSilver Scalable Multicast (QSM)," *Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (NCA 2008)*, Cambridge, MA, July 10-12, 2008.

[3] Freedman, D., Marian, T., Lee, J., Birman, K., Weatherspoon, H., and Xu, C., "Exact Temporal Characterization of 10 Gbps Optical Wide-area Network," *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC 2010)*, Melbourne, Australia, November 1-3, 2010 (Early Accept paper honor).

[4] Birman, K., Huang, Q. and Freedman, D., *"*Overcoming the "D" in CAP: Using Isis^2 To Build Locally Responsive Cloud Services," *Cornell University Technical Report*, Ithaca, NY, April, 2011.

[5] Marian, T., Freedman, D., Birman, K. and Weatherspoon, H., *"*Empirical Characterization of Uncongested Optical Lambda Networks and 10GbE Commodity Endpoints," *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, Chicago, Illinois, June 28-July 1, 2010.

[6] Birman, K., Malkhi, D. and vanRenesse, R., "Virtually Synchronous Methodology for Dynamic Service Replication," *Microsoft Research Technical Report MSR-2010-151*, November 18, 2010.

[7] Brewer, E., "Towards Robust Distributed Systems," *Keynote Presentation at the 19th ACM Symposium on Principles of Distributed Computing (PODC '00),* July 19, 2000.

[8] Gilbert, S. and Lynch, N., 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News,* Vol. 33 (2), pp. 51-59, June, 2002.

[9] Pritchett, D., "BASE: An Acid Alternative," *ACM Queue*, Vol. 6 (3), pp. 48-55, May/June, 2008.

[10] Vogels, W., "Eventually Consistent," *ACM Queue*, Vol. 6 (6), pp. 14-19, October, 2008.

[11] Lamport, L., "Paxos made simple," *ACM SIGACT News (Distributed Computing Column)*, Vol. 32 (4), pp. 51-58, December, 2008.

[12] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's highly available key-value store," *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, October 14-17, 2007.

[13] Birman, K. and Joseph, T., "Exploiting Virtual Synchrony in Distributed Systems," *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87*), Austin, TX, November 8-11, 1987.

[14] Birman, K., "A *History of the Virtual Synchrony Replication Model*," in Replication: Theory and Practice, Charron-Bost, B., Pedone, F., and Schiper, A. (Eds), LNCS, Vol. 5959, pp. 91-120, 2010.

[15] Lamport, L., "The part-time parliament," *ACM Transactions on Computer Systems (TOCS),* Vol. 16 (2), pp. 133-169, May, 1998.

[16] Gray, J., Helland, P., O'Neill, P. and Shasha, D., "The dangers of replication and a solution," *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*, Montreal, QC, Canada, pp. 173-182, June 4-6, 1996.

[17] Wada, H., Fekete, A., Zhao, L., Lee, K. and Liu, A., "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective," *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, Asilomar, CA, pp. 135-143, January 9-12, 2011.

[18] Brantner, M., Florescu, D., Graf, D., Kossmann, D. and Kraska, T., "Building a Database on S3," *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, Vancouver, BC, Canada, pp. 251-264, June 9-12, 2008.

[19] Kossman, D., "What is new in the cloud?," *Keynote Presentation at the European Conference on Computer Systems (Eurosys '11)*, Salzburg, Austria, April 10-13, 2011.

[20] Kraska, T., Hentschel, M., Alonso, G. and Kossmann, D., "Consistency Rationing in the Cloud: Pay only when it matters," *Proceedings of the Very Large Data Base Endowment (VLDB '09)*, Lyons, France, August 24-28, 2010.

[21] Abadi, D., "*Problems with CAP, and Yahoo's little known NoSQL system*," http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html .

[22] O'Neil, P.E., "The Escrow transactional method," *ACM Transactions on Database Systems (TODS)*, Vol. 11 (4), pp. 405-430, December, 1986.

[23] Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, V, Puz, N., Weaver, D., and Yerneni, R., "PNUTS: Yahoo!'s Hosted Data Serving Platform," Proceedings of the Very Large Data Base Endowment (VLDB '08), Vol. 1, No. 2, pp. 1277-1288, Auckland, New Zealand, August 23-28, 2008.

[24] Garcia-Molina, H. and Salem, K., "Sagas," *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*, San Francisco, CA, pp. 249-259, May 27-29, 1987 .

[25] Ghemawat, S., Gobioff, H., and Leung, S-T, "The Google file system," *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, pp. 29-43, October 19-22, 2003.

[26] Burrows, M., "The Chubby Lock Service for Loosely-Coupled Distributed Systems," *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, pp. 335-350, 2006.

[27] Junqueira, F.P., and Reed, B.C., "The life and times of a zookeeper," *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*, Calgary, Canada, p. 4, August 11-13, 2009.

[28] Marandi, P.J., Primi, M., Schiper, N., and Pedone, F., "Ring-Paxos: A High-Throughput Atomic Broadcast Protocol," *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, Chicago, IL, pp. 527-536, June 28-July 1, 2010.

[29] Marandi, P., Primi, M., and Pedone, F., "High Performance State-Machine Replication," *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, pp. 454-465, June 27-30, 2011.

[30] Bolosky, W.J., Bradshaw, D., Haagens, R.B., Kusters, N.P., and Li, P., "Paxos Replicated State Machines as the Basis of a High-Performance Data Store," *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, pp. 141-154, March 30- April 1, 2011..

BIBLIOGRAPHY

Our effort was led by Dr. Ken Birman, the N. Rama Rao Professor of Computer Science. Birman formulated the concepts of virtual synchrony in the early 1980s, implemented the *original* Isis system, and headed the team that used Isis to build a number of critical national-asset-level deployments. These include the core technology components of the New York Stock Exchange (the real-time communication architecture that delivers data to overhead displays, the SEC and to ticker feed companies – about 500 endpoints in all, in a fault-tolerant network of about 1000 machines), the Swiss Exchange (the entire exchange is fully replicated with a complete replica at each member bank, about 250 in total – every operation is translated to a reliable multicast and all bank systems see every quote, every trade, and do so in realtime) the French Air Traffic Control System (both for fault-tolerance in console clusters and for reliable communication of aircraft flight plans between airports; currently in use at France's five major ATC centers, but soon to expand into many other regions within Europe), the US Naval AEGIS warship command and communications system, the Microsoft Windows Clustering solution, the IBM Websphere fault-tolerance technology, the scalability solution employed by Amazon.com in its largest datacenters and scalable applications, and a tremendous range of lower profile but equally critical distributed systems and applications. These kinds of successes haven't come easily and point to the mixture of theoretical and practical work that we've focused upon (because such problems are unique precisely in their need for rigor but also for real solutions), and also in a willingness to persevere even when the commercial community has taken some other path.

Significant AFRL support in this effort also funded Dr. Daniel Freedman as a Post-Doctoral Research Associate on this work. At the time, Freedman was a serving US Army National Guard officer, in command of the forward support company for a Pennsylvania Apache Battalion. Prior to that, he deployed to Iraq as an enlisted airborne infantryman on a 4-man (later, 6-man) Long Range Surveillance team. Freedman's academic background is cross-disciplinary, with a PhD in theoretical condensed-matter physics from Cornell, followed by this post-doc in computer science. His research interests center on questions of distributed systems and communication networks, with a particular inter-disciplinary focus that reflects his diverse past. Freedman has recently accepted a position as a new faculty member in the Faculty of Electrical Engineering at the Technion — Israel Institute of Technology.

Dr. Robbert van Renesse, a Principal Research Scientist in Cornell's Department of Computer Science, was also supported in this effort. Van Renesse leads research on a range of distributed systems topics, but with particular passon for their fault tolerance and scalability aspects: Van Renesse was the principal developer of the Amoeba Distributed Operating System, commercialized by A.C.E. and used among others by the European Space Agency for monitoring experiments. At Bell Labs he helped develop the Plan 9 Operating System, now used in various commercial products as an embedded operating system and still a popular operating system for

research. Van Renesse was the principal designer and developer of the Horus Group Communication System and the principal designer of the Ensemble Group Communication System on which the group communication system of the popular IBM WebSphere product is based. Van Renesse was the principal designer and developer of the Astrolabe peer-to-peer system that was deployed on a multi-continental platform at Amazon.com and connected tens of thousands of servers. At Microsoft and EMC he has helped with the design of cluster services, search engines, and other large scale distributed services.

Further support was directed to Dr. Hakim Weatherspoon, an Assistant Professor of Computer Science, who received his PhD in Computer Science from the University of California, Berkeley. His interests include various aspects of information systems, distributed systems, network systems, and peer-to-peer systems with focus on fault-tolerance, reliability, security, and performance of Internet-scale systems, with decentralized — autonomous, federated, multi-organizational, and cooperative — control. This effort also provided limited support for Weatherspoon's post-doc, Dr. Tudor Marian. Marian received his PhD in computer science at Cornell in 2010, with research interests that span the areas of networking, distributed systems, and operating systems. Within these areas, he focuses upon performance, fault-tolerance, and reliability issues that arise within the modern datacenter, with a strong preference for constructing deployed solutions based on solid engineering. Marian is now an engineer at Google, working on their core infrastructure. Finally, additional support funded the graduate research activities of Ryan Peterson, whose interests center upon peer-assisted content distribution, which couples upload bandwidth at peers with a logically centralized coordinator that steers peers toward an efficient allocation of resources. Peterson has also, since, joined Google.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACID	Atomicity Consistency Isolation Durability
ACM	Association of Computing Machinery
AFRL	Air Force Research Laboratories
BASE	Basically Available replicated Soft state with Eventual consistency
BSD	Berkeley Standard Distribution
C2	Command and Control
CAP	Consistency Availability Partition-tolerance
CPU	Central-Processing Unit
DHT	Distributed Hash Table
DoD	Department of Defense
ESB	Enterprise Service Bus
FAA	Federal Aviation Authority
FIFO	First-In, First-Out
HPC	High-Performance Computing
IP / IPv4	Internet Protocol (version 4.0)
IPMC	Internet Protocol MultiCast
LAN	Local Area Network
MCMD	Dr. Multicast
MPI	Message-Passing Interface
.NET	[Non-acronym term for Windows' managed execution environment]
NSF	National Science Foundation
PVM	Parallel Virtual Machines
QSM	Quicksilver Scalable Multicast
O/S	Operating System
SEC	Securities and Exchange Commission
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network